Program Verification

A Formal Approach to Proving Programs Correct

Sommerville chapter 9
Section 29.2

Peter Wood

Department of Computer Science
University of Cape Town
Private Bag, RONDEBOSCH 7700
phone: +27 21 650 2666
fax: +27 21 650 3726
e-mail: ptw@cs.uct.ac.za
URL: http://www.cs.uct.ac.za/ ptw

This documentation was prepared with LaTeX and reproduced by the Printing Department, University of Cape Town from camera-ready copy supplied by the author.

Chapter 1

Introduction

It is no doubt every programmer's dream that each program he or she writes, no matter in what language, could be submitted to another program which would determine whether the given program was correct or not, thereby freeing the programmer from the laborious task of program testing. A denunciation of program testing as a means of demonstrating program correctness is given by Edsger Dijkstra, a passionate proponent of the use of formal methods in Computer Science:

> "Program testing can be used to show the presence of bugs, but never to show their absence."

Thus, the ultimate goal of program verification is to overcome this "fallacy of debugging" by providing a mechanical means (i.e. a computer program) for proving mathematically whether a given program is correct or not.

There are at least two major problems in attaining this goal. The first is that a program can only be shown to be correct with respect to a precise *specification* of what the program is supposed to do, something which is difficult to define formally for non-trivial programs. The second problem, apparently more serious than the first, is that it is not possible, in general, to write a program which checks the correctness of any program submitted to it; in other words, the problem is *undecidable*.

We shall consider each of these problems in more detail later. Before doing so, it is important to realise that the use of formal methods, such as mathematical logic, both to verify the correctness of programs and as a methodology for *constructing* correct programs is a controversial topic. To illustrate this, some comments from a recent debate on the subject are quoted below[1].

The debate arose from reaction to a talk titled "On the Cruelty of Really Teaching Computer Science" given at the ACM Computer Science Conference in February 1988 by Dijkstra. The principal theme of his talk was that "computers represent a radical novelty ... that has no precedent in our history." In his opinion, a consequence of this is that

> "... frantic efforts at hiding or denying the frighteningly unfamiliar ... have been bundled under the name "Software Engineering" ... [which] should be known as "The Doomed Discipline"."

After slating a number of Software Engineering techniques, Dijkstra gives his solution for overcoming this radical novelty:

> "I propose that we adopt for computing science VLSAL (Very Large Scale Application of Logic). ... I expect computing science to transcend its parent disciplines, mathematics and logic."

He concludes by describing his view of of an *introductory* programming course:

[1] The quotations are taken from "A Debate on Teaching Computer Science" which appeared in *Communications of the ACM*, Vol. 32, No. 12 (Dec. 1989), pp. 1397–1414.

> "On the one hand, we teach what looks like predicate calculus .... On the other hand, we teach a simple, clean, imperative programming language ..... we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification. ... Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has *not* been implemented on campus ... ."

A less radical view is taken by David Parnas who, among other things, has been an outspoken critic of the claims made by the Strategic Defense Initiative (SDI)—better known as "Star Wars"—in the United States. He takes exception to Dijkstra's dismissal of Software Engineering with some enlightening analogies with tradition engineering disciplines.

> "Most introductory engineering texts define an engineer as one who uses science and mathematics to produce useful products. ... The fact that some people write poor papers and textbooks under the title "software engineering" does not justify abandonment of the hope that, some day, programs will be produced by properly educated professional engineers. ... Good engineering programs emphasize the use of formal methods in exactly the way suggested by Dijkstra. ... However, an engineering education also teaches students to understand the limits of mathematical models."

It is in this spirit that we wish to address the subject of program verification. Although it has its limitations, there are a number of practical benefits to be gained through using its techniques, such as,

- promoting better programming discipline,
- reducing programming errors,
- reducing testing time, and
- improving documentation.

For these reasons, program verification is usually viewed as an integral component of Software Engineering.

1.1 A Motivating Scenario

The following "story" is taken from the book by Gries [4].

We have just finished writing a large program. Among other things, the program computes as intermediate results the quotient `q` and remainder `r` arising from dividing a non-negative integer `x` by a positive integer `y`. For example, with `x` = 7 and $y = 2$, the program calculates `q` = 3 (since $7 \div 2 = 3$) and `r` = 1 (since the remainder when 7 is divided by 2 is 1).

Our program appears below, with dots "..." representing the parts of the program that precede and follow the remainder-quotient calculation. The calculation is performed as given because the program will sometimes be executed on a processor that has no integer division, and portability must be maintained at all costs! The remainder-quotient calculation actually seems quite simple; since $\div$ cannot be used, we have elected to subtract divisor `y` from a copy of `x` repeatedly, keeping track of how many subtractions are made, until another subtraction would yield a negative integer.

```
...
r := x; q := 0;
WHILE r > y DO
  BEGIN r := r - y; q := q + 1 END;
...
```

We're ready to debug the program. With respect to the remainder-quotient calculation, we're smart enough to realize that the divisor should initially be greater than 0 and that upon its termination the variables should satisfy the formula

$$x = y * q + r,$$

so we add some output statements to check the calculations:

```
...
write ('dividend x =', x, 'divisor y =', y);
r := x; q := 0;
WHILE r > y DO
  BEGIN r := r - y; q := q + 1 END;
write ('y*q+r =', y*q+r);
...
```

Unfortunately, we get voluminous output because the program segment occurs in a loop, so our first test run is wasted. We try to be more selective about what we print. Actually, we need to know values only when an error is detected. Having heard of a new feature just inserted into the compiler, we decide to try it. If a Boolean expression appears within braces { and } at a point in the program, then, whenever "flow of control" reaches that point during execution, it is checked: if false, a message and a dump of the program variables are printed; if true, execution continues normally. These Boolean expressions are called *assertions*, since in effect we are asserting that they should be true when flow of control reaches them. The systems people encourage leaving assertions in the program, because they help document it.

Protests about inefficiency during production runs are swept aside by the statement that there is a switch in the computer to turn off assertion checking. Also, after some thought, we decide it may be better always to check assertions—detection of an error during production would be well worth the extra cost.

So we add assertions to the program:

```
    ...
    {y > 0}
    r := x; q := 0;
(1) WHILE r > y DO
      BEGIN r := r - y; q := q + 1 END;
    {x = y * q + r}
    ...
```

Testing now results in far less output, and we make progress. Assertion checking detects an error during a test run because y is 0 just before a remainder-quotient calculation, and it takes only four hours to find the error in the calculation of y and fix it.

But then we spend a day tracking down an error for which we received no nice false-assertion message. We finally determine that the remainder-quotient calculation resulted in

$$x = 6, y = 3, q = 1, r = 3.$$

Sure enough, both assertions in (1) are true with these values; the problem is that the remainder should be less than the divisor, and it isn't. We determine that the loop condition should be `r >= y` instead of `r > y`. If only the result assertion were strong enough—if only we had used the assertion $x = y * q + r \wedge r < y$—we would have saved a day of work! Why didn't we think of it?

We fix the error and insert the stronger assertion:

```
...
{y > 0}
r := x; q := 0;
```

```
WHILE r ≥ y DO
  BEGIN r := r - y; q := q + 1 END;
{x = y*q+r ∧ r < y}
...
```

Things go fine for a while, but one day we get incomprehensible output. It turns out that the quotient-remainder algorithm resulted in a negative remainder `r = -2`. But the remainder shouldn't be negative! And we find out that `r` was negative because initially `x` was -2. Ahhh, another error in calculating the input to the quotient-remainder algorithm—`x` isn't supposed to be negative! But we could have caught the error earlier and saved two days searching, in fact we *should* have caught it earlier; all we had to do was make the initial and final assertions for the program segment strong enough. Once more we fix an error and strengthen an assertion:

```
...
{0 ≤ x ∧ 0 < y}
r := x; q := 0;
WHILE r ≥ y DO
  BEGIN r := r - y; q := q + 1 END;
{x = y*q+r ∧ 0 ≤ r < y}
...
```

It sure would be nice to be able to invent the right assertions to use in a less *ad hoc* fashion. Why can't we think of them? Does it have to be a trial-and-error process? Part of our problem here was carelessness in specifying what the program segment was to do—we should have written the initial assertion ($0 \le x \wedge 0 < y$) and the final assertion ($x = y*q+r \wedge 0 \le r < y$) *before* writing the program segment, for they form the definition of quotient and remainder.

But what about the error we made in the condition of the while loop? Could we have prevented that from the beginning? Is there a way to prove, just from the program and assertions, that the assertions are true when flow of control reaches them? Let's see what we can do.

Just before the loop it seems that part of our result,

$$(2)\quad x = y*q+r$$

holds, since $x = r$ and $q = 0$. And from the assignments in the loop body we conclude that if (2) is true before execution of the loop body then it is true after its execution, so it will be true just before and after *every* iteration of the loop. Let's insert it as an assertion in the obvious places, and let's also make all assertions as *strong* as possible:

```
...
{0 ≤ x ∧ 0 < y}
r := x; q := 0;
{0 ≤ r ∧ 0 < y ∧ x = y*q+r}
WHILE r ≥ y DO
  BEGIN {0 ≤ r ∧ 0 < y ≤ r ∧ x = y*q+r}
    r := r - y; q := q + 1
    {0 ≤ r ∧ 0 < y ∧ x = y*q+r}
  END;
{0 ≤ r < y ∧ x = y*q+r}
...
```

Now, how can we easily determine a correct loop condition, or, given the condition, how can we prove it is correct? When the loop terminates the condition is false. Upon termination we want $r < y$, so that the complement, $r \ge y$ must be the correct loop condition. How easy that was!

It seems that if we knew how to make all assertions as strong as possible and if we learned how to reason carefully about assertions and programs, then we wouldn't make so many mistakes, we would *know* our program was correct, and we wouldn't need to debug programs at all! Hence, the days spent running test cases, looking through output and searching for errors could be spent in other ways.

1.1.1 Discussion

The story suggests that assertions, or simply Boolean expressions, are really needed in programming. But it is not enough to know how to write Boolean expressions; one needs to know how to *reason* with them: to simplify them, to prove that one follows from another, to prove that one is not true in some state, and so forth. And, later on, we will see that it is necessary to use a kind of assertion that is not part of the usual Boolean expression language of C++, Pascal or FORTRAN, the "quantified" assertion.

Knowing how to reason about assertions is one thing; knowing how to reason about *programs* is another. In the past 10 years, computer science has come a long way in the study of proving programs correct. We are reaching the point where the subject can be taught to undergraduates, or to anyone with some training in programming and the will to become proficient. More importantly, the study of program correctness proofs has led to the discovery and elucidation of methods for *developing* programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way! If the methods are practised with care, they can lead to programs that are free of errors, that take much less time to develop and debug, and that are much more easily understood (by those who have studied the subject).

Above, we mentioned that programs could be free of errors and, in a way, we implied that debugging would be unnecessary. This point needs some clarification. Even though we can become more proficient in programming, we will still make errors, even if only of a syntactic nature (typos). We are only human. Hence, some testing will always be necessary. But it should not be called debugging, for the word debugging implies the existence of bugs, which are terribly difficult to eliminate. No matter how many flies we swat, there will always be more. A disciplined method of programming should give more confidence than that! We should run test cases not to look for bugs, but to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule.

1.2 Assertions in C

Consider the following C program which determines the maximum of three given integers:

```
#include <stdio.h>
#include <assert.h>
#define TRUE 1

main()
{
   int i, j, k, m;

   printf("Please enter 3 integers\n");
   scanf("%d%d%d", &i, &j, &k);
   assert(TRUE);
   if (i <= j)
      if (j < k)
         m = k;
      else
         m = j;
   else
      if (i < k)
         m = k;
      else
         m = i;
   assert( m >= i && m > j && m >= k );
```

```
    printf("The maximum of %d, %d and %d is %d\n", i, j, k, m);
}
```

The `assert` macro is available as part of the standard C library under most operating systems. This macro verifies a program assertion by putting diagnostics into a program. The syntax is `assert(`*expression*`)`. If *expression* is false (zero), then `assert` writes the following message on the standard error output and aborts the program:

`Assertion failed:` *expression*`, file` *filename*`, line` *linenum*

The compilation of assertions into the program can be switched off by using the preprocessor option `-DNDEBUG` or control statement `#define NDEBUG` before `#include <assert.h>`.

A successful execution of the above program is given below (program output is in typewriter font, while user input is in italics).

```
Please enter 3 integers
1 2 3
The maximum of 1, 2 and 3 is 3
```

On the other hand, if we enter the integers 1, 3, 2, the following execution results.

```
Please enter 3 integers
1 3 2
Assertion failed:  m >= i && m > j && m >= k , file nonloop.c, line 22
Abort or IOT trap - core dumped
```

In this case, the program is correct while the assertion itself is wrong; the middle term should be `m >= j`. In other words, the programmer has provided an incorrect specification of what the program does. From a program maintenance point of view, this is perhaps an even more serious error than a programming error.

So assertions as supplied by the C compiler do not contribute at all to proving that a program is correct. They do, however, provide a means of specifying conditions that are expected to be true of the program variables and help in program testing.

1.3 A Simple Programming Language

For our purposes, we will consider a very simple programming language containing only assignment statements, while loops, and conditional (if-then-else) statements. This will allow us to concentrate on the principles of program correctness without getting bogged down in the detailed syntax of a full-blown programming language. At the same time, it should be noted that such a simple language still has the power to express any computable function.

The syntax of the language follows. Symbols such as v, x and y are used to denote arbitrary program *variables*, S and S_i denote program *statements*, and C and C_i denote *conditional expressions*.

Assignment Statement

$v := e$

The state (of the program variables) is changed by assigning the value of expression e to the variable v. For example,

```
x := x + 1;
```

adds one to the value of the variable x.

Sequential Composition

$S_1; S_2; \cdots S_n$

The statements $S_1, \ldots, S_n$ are executed sequentially in the given order. For example,

```
t := x; x := y; y := t;
```

exchanges the values of variables x and y using t as a temporary variable. Notice that this statement has the *side effect* of changing the value of the variable t to the old value of x.

One-Sided Conditional

IF C THEN S

If the condition C is true in the current state (of the program), then statement S is executed. If C is false, nothing is done. For example, in

```
IF (x <> 0) THEN r := y DIV x;
```

r is assigned the result of dividing the value of y by the value of x if the value of x is not zero.

Two-Sided Conditional

IF C THEN S_1 ELSE S_2

If the condition C is true in the current state (of the program), then statement S_1 is executed. If C is false, then S_2 is executed. For example, in

```
IF (x <> y) THEN max := y ELSE max := x;
```

the value of the variable max is set to the maximum of the values of x and y.

While-Loop

WHILE C DO S

If the condition C is true in the current state (of the program), then statement S is executed and the WHILE-statement is repeated. If C is false, nothing is done. Thus S is repeated executed until the value of C becomes true. If C never becomes true, then the execution of the command never terminates. For example, in

```
WHILE (x <> 0) DO x := x-2;
```

the value of x will be repeatedly decremented by 2 if the old value of x is non-zero. The statement will terminate (with x having value 0) if the original value of x is an even non-negative number. In any other state, it will not terminate.

In addition to the above syntax, the keywords BEGIN and END are used to delimit compound statements. As an example, the program to compute the quotient and remainder after dividing one integer by another given in Section 1.1 is repeated below.

```
r := x; q := 0;
WHILE r ≥ y DO
  BEGIN r := r - y; q := q + 1 END;
```

On each iteration of the loop, the compound statement `r := r - y; q := q + 1` is executed.

Having defined our programming language, we now consider how we might specify what a particular program is expected to do, in order that we can prove that the program is correct.

1.4 Program Specifications

Given a program S (for example in the language of the previous section) along with conditions P and Q defined on the variables of S, the notation

$$\{P\}\ S\ \{Q\},$$

is called a *partial correctness specification*. The condition P is called its *precondition* and condition Q is called its *postcondition*. This notation was introduced by Tony Hoare in [1].

Conditions on program variables will be written using standard mathematical notations together with logical operators like $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not') and $\Rightarrow$ ('implies'). These are described further in Chapter 2.

We say that $\{P\}\ S\ \{Q\}$ is true, if whenever S is executed in a state satisfying P and if the execution of S terminates, then the state in which S's execution terminates satisfies Q.

Example 1.4.1 Consider the specification

$$\{x = 1\}\mathtt{x := x + 1}\{x = 2\}.$$

Here P is the condition that the value of x is 1, Q is the condition that the value of x is 2, and S is the assignment statement `x := x+1`. Specification $\{x = 1\}$ `x := x+1` $\{x = 2\}$ is clearly true. □

These specifications are 'partial' because for $\{P\}\ S\ \{Q\}$ to be true it is not necessary for the execution of S to terminate when started in a state satisfying P. It is only required that if the execution terminates, then Q holds.

A stronger kind of specification is a total correctness specification. There is no standard notation for such specifications. We shall use $[P]\,S\,[Q]$. A total correctness specification $[P]\,S\,[Q]$ is true if and only if the following conditions apply:

1. Whenever S is executed in a state satisfying P, then the execution of S terminates.

2. After termination, Q holds.

The relationship between partial and total correctness can be informally expressed by the equation:

Total correctness = Termination + Partial correctness.

Total correctness is what we are ultimately interested in, but it is usually easier to prove it by establishing partial correctness and termination separately.

Termination is often straightforward to establish, but there are some well-known examples where it is not. For example, no one knows whether the program below terminates for all values of x:

```
WHILE x > 1 DO
  IF ODD (x) THEN x := (3 × x)+1 ELSE x := x DIV 2
```

(The expression `x DIV 2` evaluates to the result of rounding down to $x/2$ to a whole number).

We spend most of our time dealing only with partial correctness. Theories of total correctness can be found in the texts by Dijkstra [7] and Gries [4].

1.4.1 Some examples

The examples below illustrate various aspects of partial correctness specification.

In Examples 5, 6 and 7 below, "*true*" is the condition that is always true. In Examples 3, 4 and 7, "$\wedge$" is the logical operator 'and', i.e. if P_1 and P_2 are conditions, then $P_1 \wedge P_2$ is the condition that is true whenever both P_1 and P_2 hold.

1. $\{x = 1\}$ `y := x` $\{y = 1\}$

This says that if the statement `y := x` is executed in a state satisfying the condition $x = 1$ (i.e. a state in which the value of x is 1), then, if the execution terminates (which it does), then the condition $y = 1$ will hold. Clearly this specification is true.

2. $\{x = 1\}$ `y := x` $\{y = 2\}$

This says that if the execution of `y := x` terminates when started in a state satisfying $x = 1$, then $y = 2$ will hold. This is clearly false.

3. $\{x = x_0 \wedge y = y_0\}$ `BEGIN t := x; x := y; y := t END` $\{x = y_0 \wedge y = x_0\}$

This says that if the execution of `BEGIN t := x; x := y; y := t END` terminates (which it does), then the values of x and y are exchanged. The variables x_0 and y_0, which don't occur in the statement and are used to name the initial values of program variables x and y, are called *auxiliary* variables (or *ghost* variables).

4. $\{x = x_0 \wedge y = y_0\}$ `BEGIN x := y; y := x END` $\{x = y_0 \wedge y = x_0\}$

This says that `BEGIN x := y; y := x END` exchanges the values of x and y. This is not true.

5. $\{true\}\ S\ \{Q\}$

This says that whenever S halts, Q holds.

6. $\{P\}\ S\ \{true\}$

This specification is true for every condition P and every statement S (because *true* is always true).

7. $\{true\}$
```
BEGIN
  r := x;
  q := 0;
  WHILE y ≤ r DO
    BEGIN r := r - y; q := q + 1 END
END
```
$\{r < y \wedge x = r + (y \times q)\}$

This specification is $\{true\}\ S\ \{r < y \wedge x = r + (y \times q)\}$ where S is the compound statement above. The specification is true if whenever the execution of S halts, then q is the quotient and r is the remainder resulting from dividing y into x. It is true (even if x is initially negative). Compare this to the example in Section 1.1.

In this example a program variable q is used. This should not be confused with the Q used in 5 above. The program variable q ranges over numbers, whereas the postcondition Q ranges over conditions. In general, we use lower-case letters for particular program variables and upper-case letters in *italic font* for variables ranging over conditions. Preconditions, postconditions and assertions in general will be in italic font, whereas program statements will be in `typewriter font`. Although this subtle use of fonts might appear confusing at first, once you get the hang of things the difference between variables and statements will be clear (indeed you should be able to disambiguate things from context without even having to look at the font).

Chapter 2

A Taste of Logic

In order to make statements about conditions on the program variables that we believe to be true at various points in a program, as well as to be able to deduce what other conditions must be true as a result, we need to endure a brief excursion into some of the formalities of *predicate calculus*. Before doing so, however, we consider the simpler system of *propositional calculus*.

2.1 The Propositional Calculus

Propositional calculus is a simple language for expressing formal proofs. Its primary constituents are propositions, that is, sentences that are either true or false; there are no variables in a proposition that make it true for some values of the variables and false for others. The propositional calculus is essentially a formalisation of the logical operators tabulated below.

$\neg$	not (negation)
$\wedge$	and (conjunction)
$\vee$	or (disjunction)
$\Rightarrow$	implies (if ... then)
$\Leftrightarrow$	if and only if

We introduce some of the terminology by means of an example in English. Consider the following argument.

1. If Superman were able and willing to prevent evil, he would do so.
2. If he were unable to prevent evil, he would be impotent.
3. If he were unwilling to prevent evil, he would be malevolent.
4. Superman does not prevent evil.
5. If Superman exists, he is neither impotent nor malevolent.
6. Therefore Superman does not exist.

Items (1) to (5) above are called the *premises* of the argument, while (6) is the *conclusion*. We are concerned with whether or not the conclusion *logically follows* from the premises. If so, the argument is *logical* or *valid*.

We can decompose the argument into propositions which are combined by means of the logical connectives given above. The individual propositions, each assigned a unique symbol, are as follows:

X	Superman exists.
W	Superman is willing to prevent evil.
A	Superman is able to prevent evil.
M	Superman is malevolent.
I	Superman is impotent.
E	Superman prevents evil.

In terms of these propositions and the logical connectives, the argument becomes:

$($	
$((W \wedge A) \Rightarrow E)$	If Superman were willing and able to prevent evil, he would do so.
$\wedge$	
$((\neg A) \Rightarrow I)$	If he were unable to prevent evil, he would be impotent.
$\wedge$	
$((\neg W) \Rightarrow M)$	If he were unwilling to prevent evil, he would be malevolent.
$\wedge$	
$(\neg E)$	Superman does not prevent evil.
$\wedge$	
$(X \Rightarrow \neg(I \vee M))$	If Superman exists, he is neither impotent nor malevolent.
$)$	
$\Rightarrow$	Therefore,
$(\neg X)$	Superman does not exist.

An argument is valid if it is impossible for the premises to be true and the conclusion to be false.

Suppose P and Q are propositions; then:

$\neg P$	is true if P is false, and false if P is true.
$P \wedge Q$	is true whenever both P and Q are true.
$P \vee Q$	is true if either P or Q (or both) are true.
$P \Rightarrow Q$	is true if whenever P is true, then Q is true also. By convention we regard $P \Rightarrow Q$ as being true if P is false. In fact, it is common to regard $P \Rightarrow Q$ as equivalent to $\neg P \vee Q$; however, some philosophers called intuitionists disagree with this treatment of implication.
$P \Leftrightarrow Q$	is true if P and Q are either both true or both false. In fact $P \Leftrightarrow Q$ is equivalent to $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

A statement of the form 'if A then B', written $A \Rightarrow B$, is called an *implication*; A is called the *antecedent* and B the *consequent*.

2.1.1 Tautologies and Counterexamples

Let us consider the following valid argument:

> If an algorithm is proven, then it is reliable. Therefore, an algorithm cannot be both proven and unreliable.

If we let P denote the proposition that an algorithm is proven, and R denote the proposition that an algorithm is reliable, then the argument can be formalised as

$$(P \Rightarrow R) \Rightarrow \neg(P \wedge \neg R)$$

We can verify that the argument is valid by constructing a truth table for it as follows:

P	R	$(P \Rightarrow R)$	$\Rightarrow$	$\neg$	$(P$	$\wedge$	$\neg R)$
T	T	T	T	T	T	F	F
F	T	T	T	T	F	F	F
T	F	F	T	F	T	T	T
F	F	T	T	T	F	F	T

The implication in the argument is true whatever the values of P and R. A *tautology* is a propositional form that is true whatever assignment of true or false is given to each of its constituent simple propositions. An *argument* is an implication in which the premises form the antecedent and the conclusion forms the consequent. Now we see that an argument is *valid* if and only if it is a tautology.

The following is an example of an invalid argument:

> If an algorithm is proven, then it is reliable. Therefore, an algorithm is proven or it is not reliable.

This argument can be formalised as

$$(P \Rightarrow R) \Rightarrow (P \vee \neg R)$$

Once again we can construct a truth table, this time to show that the argument is not valid.

P	R	$(P \Rightarrow R)$	$\Rightarrow$	$(P$	$\vee$	$\neg R)$
T	T	T	T	T	T	F
F	T	T	F	F	F	F
T	F	F	T	T	T	T
F	F	T	T	F	T	T

A *counterexample* to the argument is given by P being false and R being true.

2.1.2 Equivalences

Truth tables work well for establishing the validity of arguments if the number of simple propositions is small; otherwise, they are impractical as the number of cases to consider grows exponentially with respect to the number of propositions. Two alternative techniques for manipulating logical formulas are the use of known equivalences between formulas, and the application of rules of inference.

Consider the two equivalent definitions of a valid argument: $(P \Rightarrow Q)$ and $\neg(P \wedge \neg Q)$. In other words,

$$(P \Rightarrow Q) \Leftrightarrow \neg(P \wedge \neg Q)$$

is a tautology. We call a tautology of the form $P \Leftrightarrow Q$ an *equivalence*, written $P \equiv Q$.

There are many equivalences between formulas in logic; some of the common ones are given below.

1. *Constants*
   $(P \vee true) \equiv true$ $(P \vee false) \equiv P$
   $(P \wedge true) \equiv P$ $(P \wedge false) \equiv false$
   $(true \Rightarrow P) \equiv P$ $(false \Rightarrow P) \equiv true$
   $(P \Rightarrow true) \equiv true$ $(P \Rightarrow false) \equiv \neg P$
2. *Law of Excluded Middle*
   $P \vee (\neg P) \equiv true$
3. *Law of Contradiction*
   $P \wedge (\neg P) \equiv false$

4. *Negation*

$$\neg\neg P \equiv P$$

5. *Idempotency*

$$(P \vee P) \equiv P$$
$$(P \wedge P) \equiv P$$

6. *Implication*

$$(P \Rightarrow Q) \equiv (\neg P \vee Q)$$
$$(P \Rightarrow Q) \equiv (\neg Q \Rightarrow \neg P)$$

7. *Associativity*

$$P \vee (Q \vee R) \equiv (P \vee Q) \vee R$$
$$P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$$

8. *Commutativity*

$$(P \wedge Q) \equiv (Q \wedge P)$$
$$(P \vee Q) \equiv (Q \vee P)$$

9. *Distributivity*

$$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$$
$$P \Rightarrow (Q \vee R) \equiv (P \Rightarrow Q) \vee (P \Rightarrow R)$$
$$P \Rightarrow (Q \wedge R) \equiv (P \Rightarrow Q) \wedge (P \Rightarrow R)$$
$$P \vee (Q \Rightarrow R) \equiv (P \vee Q) \Rightarrow (P \vee R)$$

10. *De Morgan's Laws*

$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$

2.1.3 Rules of Inference

In order to produce a formal proof that something is true, one typically needs to be able to deduce new propositions from existing ones in a sound way. This is done by means of rules of inference. If the conclusion of an argument can be derived from the premises by a sequence of steps using equivalences and rules of inference, then the argument is guaranteed to be valid.

One convention for depicting a rule of inference is to list the premises of the rule above a dividing line which has the conclusion or *inference* below it. Consider the rule

$$\frac{P \quad Q}{P \wedge Q}$$

which says that if it is possible to prove P and it is possible to prove Q, then it is valid to infer the conjunction $P \wedge Q$. This is known as the $\wedge$-introduction rule. Another well-used rule of inference is the $\Rightarrow$-elimination rule, more commonly known as *modus ponens*,

$$\frac{P \quad P \Rightarrow Q}{Q}$$

which states that Q follows from a proof of P and a proof of $P \Rightarrow Q$. A complete set of introduction and elimination rules is given below.

Introduction rules		*Elimination rules*	
∧-I:	$\dfrac{P \quad Q}{P \wedge Q}$	∧-E:	$\dfrac{P \wedge Q}{P} \quad \dfrac{P \wedge Q}{Q}$
∨-I:	$\dfrac{P}{P \vee Q} \quad \dfrac{Q}{P \vee Q}$	∨-E:	$\dfrac{P \vee Q \quad \begin{matrix}[P] \\ R\end{matrix} \quad \begin{matrix}[Q] \\ R\end{matrix}}{R}$
⇒-I:	$\dfrac{\begin{matrix}[P] \\ Q\end{matrix}}{P \Rightarrow Q}$	⇒-E:	$\dfrac{P \quad P \Rightarrow Q}{Q}$
¬-I:	$\dfrac{\begin{matrix}[P] \\ false\end{matrix}}{\neg P}$	¬-E:	$\dfrac{P \quad \neg P}{false} \quad \dfrac{false}{P}$

The notation $[P]$ is read as "assuming P is true" so that the ∨-elimination rule reads as follows: if only two cases P and Q need be considered, and if R follows from assuming P and also from assuming Q, then it is valid to infer R in all cases.

Example 2.1.1 Let us now consider a complete *formal* proof, in this case one direction of one of the distributivity laws. We will prove

$$P \wedge (Q \vee R) \Rightarrow (P \wedge Q) \vee (P \wedge R)$$

The proof proceeds as follows.

Assume	1. $P \wedge (Q \vee R)$	
	2. P	(1, ∧-E)
	3. $Q \vee R$	(1, ∧-I)
	Assume 4. Q	
	5. $P \wedge Q$	(2, 4, ∧-I)
	6. $(P \wedge Q) \vee (P \wedge R)$	(5, ∨-I)
	Assume 7. R	
	8. $P \wedge R$	(2, 7, ∧-I)
	9. $(P \wedge Q) \vee (P \wedge R)$	(8, ∨-I)
	10. $(P \wedge Q) \vee (P \wedge R)$	(3, 4, 6, 7, 9, ∨-E)
	11. $P \wedge (Q \vee R) \Rightarrow (P \wedge Q) \vee (P \wedge R)$	(1,10, ⇒-I)

□

2.2 The Predicate Calculus

For simplicity, only a fragment of this language will be used. Things like:

$$true, \quad false, \quad x = 1, \quad r < y, \quad x = r + (y \times q)$$

are examples of *atomic statements*. Statements are either true or false. The statement *true* is always true and the statement *false* is always false. The statement $x = 1$ is true if the value of x is equal to 1. The statement $x < y$ is true if the value of x is less than the value of y. The statement $x = r + (y \times q)$ is true if the value of x is equal to the sum of the value of r with the product of y and q.

Thus, the predicate calculus deals with relative truths, that is, statements whose truth depends on the values of variables such as x and y. We say that $x < y$ is *predicated on* x and y, or the relation $x < y$ is a *predicate*.

Statements are built out of *terms* like:

$$x, \quad 1, \quad r, \quad y, \quad r + (y \times q), \quad y \times q$$

Terms denote *values* such as numbers and strings, unlike statements which are either true or false. Some terms, like 1 and 4 + 5, denote a fixed value, whilst other terms contain variables like x, y, z, etc. whose value can vary. We will use conventional mathematical notation for terms, as illustrated by the examples below:

$$\begin{array}{ccc} x, & y, & z, \\ 1, & 2, & 325, \\ -x, & -(x+1), & (x \times y) + z, \\ \sqrt{(1+x)}, & x!, & sin(x) \end{array}$$

The statements *true* and *false* are atomic statements that are always true and false respectively. Other atomic statements are built from terms using *predicates*. Here are some more examples:

$$odd(x), \quad prime(3), \quad x = 1, \quad (x+1)^2 \geq x^2$$

The predicates above are *odd* and *prime*, along with $=$ and $\geq$ which are examples of *infixed* predicates. The expressions x, 1, 3, $x + 1$, $(x + 1)^2$, x^2 are examples of terms.

Compound statements are built up from atomic statements using the same logical operators as in the propositional calculus. Examples of statements built using the connectives are:

$odd(x) \vee even(x)$	x is odd or even.
$\neg(prime(x) \Rightarrow odd(x))$	It is not the case that if x is prime, then x is odd.
$x \leq y \Rightarrow x \leq y^2$	If x is less than equal or equal to y, then x is less than or equal to y^2.

To reduce the need for brackets it is assumed that $\neg$ binds more strongly than $\wedge$ and $\vee$, which in turn bind more strongly than $\Rightarrow$ and $\Leftrightarrow$. For example:

$$\begin{array}{rll} \neg P \wedge Q & \text{is equivalent to} & (\neg P) \wedge Q \\ P \wedge Q \Rightarrow R & \text{is equivalent to} & (P \wedge Q) \Rightarrow R \\ P \wedge Q \Leftrightarrow \neg R \vee S & \text{is equivalent to} & (P \wedge Q) \Leftrightarrow ((\neg R) \vee S) \end{array}$$

The value of a variable x is called the *state* of x, for example, x might have value 1. The set of all possible values for x, such as the set of integers, for example, is called the *state space* of x. From now on we will assume that all variables range over the integers unless otherwise stated. The state of a program is the state of all its variables. When a predicate is true in a given state, we say that the state *satisfies* the predicate.

Example 2.2.1 The program state given by ($i = 2$, $j = 3$) satisfies the predicate $i < j$, while the state ($i = 2$, $j = 2$) does not. □

When *no* state satisfies predicate P, then P is said to be *unsatisfiable*. On the other hand, if *all* states satisfy P, P is said to be *valid*.

Example 2.2.2 The statement $(i < j) \wedge (j < i)$ is unsatisfiable since there are no two integers for which it is true. The statement $(i < j) \Rightarrow (i \leq j)$ is true for all integers i and j; it is therefore valid. □

2.2.1 Set notation

We will often find it useful to use set-former notation which is of the general form

$$\{\text{variables} : \text{predicate}\}.$$

Example 2.2.3 The set

$$\{i : i \geq 0\}$$

is the set of all integers i such that i is at least 0. The set

$$\{i, j : i < j\}$$

is the set of pairs of integers i, j such that i is less than j. □

These sets identify a subset of a *universe* of values, for example, the set of all integers, or the set of all pairs of integers. The universe of integers can be written as

$$\{i : true\},$$

while

$$\{i : false\} = \emptyset.$$

If P is predicated on i, then we can show that $\{i : P(i)\}$ is empty by showing that $P(i)$ is unsatisfiable, that is, $P(i) \equiv false$.

2.2.2 Logical Quantifiers

We would like to be able to make statements such as "every element in the set has the property that . . ." or "there is some element in the set having the property . . .." In order to do so, we need operators called *quantifiers*.

Universal quantifiers

The universal quantifier is denoted by $\forall$ and is pronounced "for all." Consider the statement

$$\forall(i : 0 \leq i < n : a[i] \geq 0).$$

This says that for all (integers) i, such that i is between 0 and $n-1$, $a[i]$ is positive. The bracketed expression describes a set followed by a predicate on that set. The variable i is a *bound* variable and the set $\{i : 0 \leq i < n\}$ is the *range* of the bound variable.

Example 2.2.4 The statement

$$\forall(i : 0 < i < n : a[i-1] \leq a[i])$$

states that the elements of array a are stored in increasing order. □

The range of a bound variable is omitted if it is *true*. For example, in

$$\forall(i :: i \times 0 = 0)$$

the multiplication of an arbitrary value by 0 always yields 0. As before, the type of i is implicitly assumed to be integer.

Existential quantifiers

The existential quantifier is denoted by $\exists$ and is pronounced "there exists." Consider the statement

$$\exists(i : 0 \leq i < n : a[i] = x).$$

This says that there exists an (integer) i, such that i is between 0 and $n - 1$ and $a[i]$ equals x. As with $\forall$, there are two forms of $\exists$:

$$\exists(v : R : P)$$

read "there exists a v in the range R such that P, and

$$\exists(v :: P)$$

read "there exists a v such that P," where the range of v is implicitly the type of v. Note that the equivalences

$$\exists(v : R : P) \equiv \neg\forall(v : R : \neg P)$$

and

$$\forall(v : R : P) \equiv \neg\exists(v : R : \neg P)$$

exists between sentences involving quantifiers.

We have already seen an example of a bound variable. For instance, in $Q(i : R(i) : P(i))$, variable i is bound to the quantifier Q. Those occurrences of a variable not bound to a quantifier in some expression R are said to be *free* in R. For example, in

$$\exists(r :: x = y \times r)$$

r is bound while x and y are free.

$\vdash$ means it is provable

$\models$ means it is true

Chapter 3

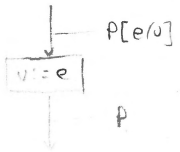
Proof Rules for Programs

In this chapter we will be concerned with how to prove that a given program is correct with respect to its specifications. The proofs that we generate will be formal proofs based on manipulating assertions by means of rules of inference. We will use rules of inference that are specific to the constructs in our programming language, one rule per construct. These rules were originally proposed by Hoare in [1].

Since the proofs we generate are based essentially on the manipulation of symbols according to fixed rules, they can become long and tedious. On the other hand, this also means that such proofs can be mechanised to a large extent, so that the tiresome details can be left to a computer program to figure out. Our reason for covering the rules is so that one can appreciate the logical foundations of program verification systems, and that for describing a number of examples in detail is so that one can see the steps such a verifier would have to take.

3.1 The Assignment Statement

The simplest proof rule for programs is that associated with assignment statements. It is referred to as an *axiom* because it does not depend on anything else being proved, that is, it is always true for any assignment statement.



Assignment Axiom

$$\vdash \{P[e/v]\}\ v := e\ \{P\}$$

In the assignment axiom, v is a variable, e is an expression, and the notation $\{P[e/v]\}$ means that e is substituted for v wherever it occurs in P. Notice that this implies that one usually works backwards from P to $P[e/v]$.

Example 3.1.1 A couple of trivial examples follow:

$$\{y > 0\}\ \mathrm{x} := \mathrm{y}\ \{x > 0\}$$

$$\{x \geq 0\}\ \mathrm{x} := \mathrm{x} + 1\ \{x > 0\}$$

Note that in the second example we have used the fact that $x + 1 > 0 \equiv x \geq 0$. □

The following two rules are used to manipulate specifications, allowing us to match up parts of a proof of correctness (as we shall demonstrate later). Essentially the rules say that we can always strenthen the precondition and weaken the postcondition of a specification.

Precondition Stengthening

$$\frac{\vdash P \Rightarrow P' \qquad \vdash \{P'\}\ S\ \{Q\}}{\vdash \{P\}\ S\ \{Q\}}$$

Precondition P is stronger than precondition P', so P describes *fewer* states than P': any state satisfying P also satisfies P'. So if $\{P'\}\ S\ \{Q\}$ is valid and $P \Rightarrow P'$, then it is valid to infer $\{P\}\ S\ \{Q\}$. The analogous rule for postconditions is *postcondition weakening*.

Postcondition Weakening

$$\frac{\vdash \{P\}\ S\ \{Q'\} \qquad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ S\ \{Q\}}$$

Postcondition Q is weaker than postcondition Q', so Q describes *more* states than Q': any state satisfying Q' also satisfies Q. In some texts, the above two rules are called the *consequence rules*.

Example 3.1.2 Consider the specification

$$\{i \geq 0\}\ \mathtt{i} := \mathtt{i} + 1\ \{i > 0\}.$$

The precondition $\{i > 0\}$ is stronger than $\{i \geq 0\}$, so using the precondition strengthening rule we can derive the specification

$$\{i > 0\}\ \mathtt{i} := \mathtt{i} + 1\ \{i > 0\}.$$

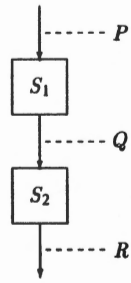
Since $\{i \geq 0\}$ is weaker than $\{i > 0\}$, we could also derive

$$\{i \geq 0\}\ \mathtt{i} := \mathtt{i} + 1\ \{i \geq 0\}$$

by the postcondition weakening rule. □

3.2 Sequences of Statements

The next rule, the *rule of sequential composition*, is used for combining specifications about individual program statements into specifications for sequences of statements.



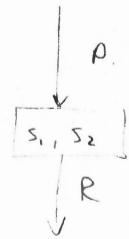


Figure 3.1: The rule of sequential composition

Proof Rule For Sequential Composition

$$\frac{\vdash \{P\}\, S_1\, \{Q\} \qquad \vdash \{Q\}\, S_2\, \{R\}}{\vdash \{P\}\, S_1; S_2\, \{R\}}$$

To paraphrase the rule: if executing S_1 with precondition P guarantees postcondition Q, and executing S_2 with precondition Q guarantees postcondition R, then executing $S_1; S_2$ with precondition P guarantees postcondition R. Diagrammatically this can be visualised as shown in Figure 3.1.

Example 3.2.1 Let us give a formal proof of correctness of our routine for exchanging the values of variables x and y.

$\{x = x_0 \wedge y = y_0\}$
`BEGIN t := x; x := y; y := t END`
$\{x = y_0 \wedge y = x_0\}$

Working backwards, the proof proceeds as follows. First we obtain a precondition for `y := t` using the supplied postcondition along with the assignment axiom:

$\{t = x_0 \wedge x = y_0\}$
`y := t`
$\{x = y_0 \wedge y = x_0\}$

Next we use this precondition as the postcondition for `x := y`, and once again apply the assignment axiom to obtain a precondition for this statement:

$\{t = x_0 \wedge y = y_0\}$
`x := y;`
$\{t = x_0 \wedge x = y_0\}$

Now we can combine these two results using the rule of sequential composition:

$\{t = x_0 \wedge y = y_0\}$
`BEGIN x := y; y := t END`
$\{x = y_0 \wedge y = x_0\}$

Using the above precondition as the postcondition for `t := x`, we obtain the precondition for this statement using the assignment axiom:

$\{x = x_0 \wedge y = y_0\}$
`t := x;`
$\{y = y_0 \wedge t = x_0\}$

Since this precondition is the same as the original, the result follows from the above two steps by one more application of the rule of sequential composition:

$\{x = x_0 \wedge y = y_0\}$
`BEGIN t := x; x := y; y := t END`
$\{x = y_0 \wedge y = x_0\}$

The may seem like an extremely laborious way of proving a trivial result. Nevertheless, as we have stated before, these are the kinds of steps an automated verifier would have to take. In addition, as a result of going through the above, we will be better equipped to tackle less obvious results. □

3.3 Conditional Statements

We have now seen two proof rules: the assignment axiom and the rule of sequential composition. In order to prove results about programs in our simple language that do not contain loops, we need to have proof rules dealing with conditional statements. Since there are two types of such statements, we have two proof rules, a *rule for one-sided conditionals* and a *rule for two-sided conditionals*.

Proof Rule For One-Sided Conditionals

$$\frac{\vdash \{P \wedge B\}\ S\ \{Q\} \qquad \vdash P \wedge \neg B \Rightarrow Q}{\vdash \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ S\ \{Q\}}$$

This rule can be paraphrased as follows: if executing S with precondition $P \wedge B$ guarantees postcondition Q, and it can be proved that $P \wedge \neg B \Rightarrow Q$, then executing `IF` B `THEN` S with precondition P guarantees postcondition Q. In other words, we have to show that if we start with the program in a state satisfying P, then whether we execute S or not the program ends up in a state satisfying Q. Diagrammatically this can be represented as shown in Figure 3.2.

The proof rule for two-sided conditional statements is similar of course, except that now we have to consider statement S_2 when B is false.

Proof Rule For Two-Sided Conditionals

$$\frac{\vdash \{P \wedge B\}\ S_1\ \{Q\} \qquad \vdash \{P \wedge \neg B\}\ S_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \{Q\}}$$

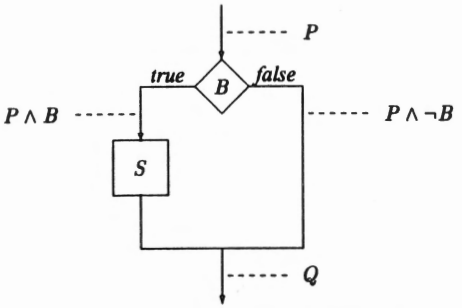


Figure 3.2: The rule for one-sided conditionals

This rule can be paraphrased as follows: if executing S_1 with precondition $P \wedge B$ guarantees postcondition Q, and executing S_2 with precondition $P \wedge \neg B$ guarantees postcondition Q, then executing `IF` B `THEN` S_1 `ELSE` S_2 with precondition P guarantees postcondition Q. In other words, we have to show that if we start with the program in a state satisfying P, then no matter which part of the conditional statement is executed the program ends up in a state satisfying Q. Diagrammatically this can be represented as shown in Figure 3.3.

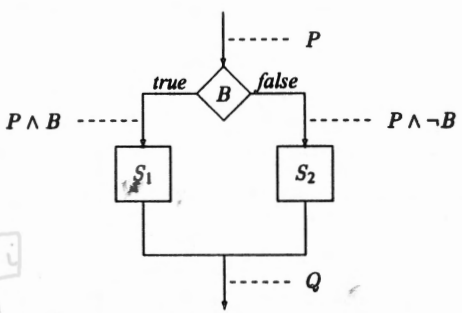


Figure 3.3: The rule for two-sided conditionals

Example 3.3.1 Let us now consider an example in which we need to use these proof rules, namely the example from Section 1.2 which determines the maximum of three integers.

```
{true}
IF i <= j
  THEN IF j < k THEN m := k ELSE m := j
  ELSE IF j < k THEN m := k ELSE m := i
{m ≥ i ∧ m ≥ j ∧ m ≥ k}
```

The following is a formal proof of correctness of the above program fragment. For notational convenience, let us denote the first nested `IF`-statement as S_1 and the second as S_2. Thus, the above statement is `IF i <= j THEN` S_1 `ELSE` S_2. It makes sense to decompose the proof into three parts: (1) find a precondition for S_1, (2) find a precondition for S_2, and (3) prove the overall specification using the rule for two-sided conditionals.

Clearly, S_1 is only executed when $i \leq j$, so we begin by attempting to prove

$$\{i \leq j\}\ S_1\ \{m \geq i \wedge m \geq j \wedge m \geq k\}.$$

In order to prove this, we will need to work backwards finding preconditions for each of the components of the conditional statement S_1, and then use the rule for two-sided conditionals. First, we obtain a precondition for `m := k` using the assignment axiom.

$$\{k \geq i \wedge k \geq j\}\ \texttt{m := k}\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{1}$$

Then using the assignment axiom again we obtain a precondition for the `ELSE`-part of S_1.

$$\{j \geq i \wedge j \geq k\}\ \texttt{m := j}\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{2}$$

As an aside, notice that if we were using the incorrect postcondition of Section 1.2 in which $m > j$, we would derive an unsatisfiable precondition at this point, one in which $j > j$. This would immediately tell us that our postcondition was incorrect.

In order to have the assertions in the correct form to apply the rule for two-sided conditionals, we will have to strengthen each of the preconditions in (1) and (2). Using the normal rules of inequalities, we have the following two implications.

$$(i \leq j \wedge j < k) \Rightarrow (k \geq i \wedge k \geq j) \tag{3}$$
$$(i \leq j \wedge j \geq k) \Rightarrow (j \geq i \wedge j \geq k) \tag{4}$$

We now apply the precondition strengthening rule to (1,3) and (2,4), thereby deriving the following two specifications.

$$\{i \leq j \wedge j < k\}\ \texttt{m := k}\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{5}$$
$$\{i \leq j \wedge j \geq k\}\ \texttt{m := j}\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{6}$$

Now (5) and (6) are in the form in which the two-sided conditional rule can be applied directly to obtain the desired result.

$$\{i \leq j\}\ S_1\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{7}$$

Following steps similar to those above, we can derive an analogous proof for a valid precondition for S_2.

$$\{j \leq i\}\ S_2\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{8}$$

We cannot apply the two-sided conditional rule directly to (7) and (8) because $\{j \leq i\}$ is not the negation of $\{i \leq j\}$. Nevertheless we can modify (8) by using the fact that $j < i \Rightarrow j \leq i$ and precondition stengthening to yield the following.

$$\{j < i\}\ S_2\ \{m \geq i \wedge m \geq j \wedge m \geq k\} \tag{9}$$

Now applying the two-sided conditional rule to (7) and (9) proves that our original program fragment is correct with respect to its pre- and postconditions. □

3.4 While Statements

In order to prove the correctness of a `WHILE`-statement, we need to find an assertion that is *invariant*, that is, the assertion is true each time around the loop. This is very similar to finding the correct inductive hypothesis for a proof by mathematical induction. In fact, the similarity extends further in that just as finding the correct inductive hypothesis is usually the most difficult part of a proof by induction, finding a strong enough loop invariant is usually much harder than the subsequent proof of correctness of the loop. For this reason, most program verification systems require that the user supply an invariant for each loop in a program, after which the system will ensure that each is indeed invariant and will attempt to complete the proof of correctness.

Example 3.4.1 Consider the following program which adds the sum of a sequence of integers to a variable s.

```
WHILE (i <> n) DO
  BEGIN
    s := s + i;
    i := i + 1
  END
```

An invariant I for the above loop is $s = i * (i - 1)/2$, which looks rather similar to the formula one would use in the inductive hypothesis in a proof by induction of the formula for the sum of an arithmetic progression. The fact that I is an invariant means that if I is true before the body of the loop, say S, is executed, then it is true afterwards. In other words, $\{I\}\ S\ \{I\}$ is a correct specification. □

Up until now, all programs we have considered have been guaranteed to terminate since they contained no loops. This means that proofs of partial correctness were all that were required; partial correctness always implied total correctness. Now that we are considering loops, the situation changes. For instance, the program given in Example 3.4.1 above will not terminate if i starts off being larger than n. The proof rule for WHILE-statements we consider is one for partial correctness, that is, it assumes that the given loop terminates. A separate proof of termination is necessary to show the total correctness of each loop. We first consider the proof rule for partial correctness.

Proof Rule For While Statements

$$\frac{\vdash \{I \wedge B\}\ S\ \{I\}}{\vdash \{I\}\ \texttt{WHILE}\ B\ \texttt{DO}\ S\{I \wedge \neg B\}}$$

This rule states that if we can prove that I is left unchanged by a single execution of S (in which case B must also be true in order for S to be executed), then it is unchanged after the while loop terminates (in which case B will be false).

We are more likely to find ourselves facing the task of proving that $\{P\}$ WHILE B DO S $\{Q\}$ is valid, where P and Q are general assertions. In this case, we have to (1) find an invariant I for the loop, (2) show that $P \Rightarrow I$, and (3) show that $I \wedge \neg B \Rightarrow Q$. Then the desired result follows from precondition strengthening, postcondition weakening, and the proof rule for while statements. Effectively, we have derived another proof rule for while statements.

Derived Proof Rule For While Statements

$$\frac{\begin{array}{c}\vdash P \Rightarrow I \\ \vdash \{I \wedge B\}\ S\ \{I\} \\ \vdash I \wedge \neg B \Rightarrow Q\end{array}}{\vdash \{P\}\ \texttt{WHILE}\ B\ \texttt{DO}\ S\{Q\}}$$

The first premise states that I is true before the loop executes (similar to the basis in mathematical induction), the second states that I is an invariant (similar to the induction step in mathematical induction), and the third states that, on termination of the loop (when I is still true and B must be false), Q is true. A visual representation of this rule is shown in Figure 3.4.

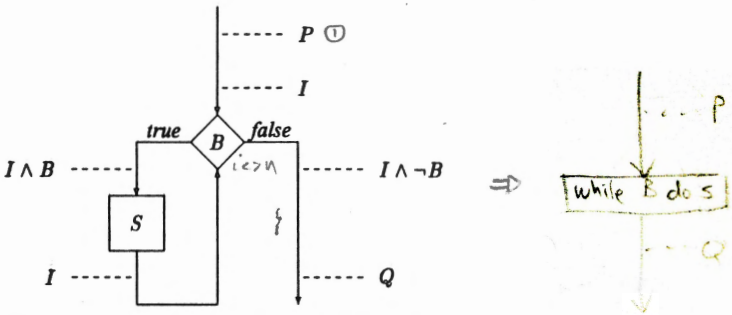


Figure 3.4: The derived rule for while statements

Example 3.4.2 Consider the while loop given in Example 3.4.1 and repeated below.

```
WHILE (i <> n) DO
  BEGIN s := s + i; i := i + 1 END
```

Suppose that we want to prove that the loop is partially correct with respect to the precondition $P \equiv (i = 0 \wedge s = 0)$ and the postcondition $Q \equiv (s = n * (n-1)/2)$. There are four steps to follow before we can use the derived proof rule for while statements. First, we think up a loop invariant I, which in this case is $s = i * (i-1)/2$. Next, we prove that I is true before the first iteration of the loop, that is, we prove that $P \Rightarrow I$. This is straightforward since if P is true, both i and s are zero and I reduces to $0 = 0$ which is clearly true.

Thirdly, we have to show that I is indeed an invariant for the loop by proving that $\{I \wedge B\}$ `s := s + i; i := i + 1` $\{I\}$. In other words, if the program is in a state satisfying I (and B if the body of the loop is to be executed) before execution of the body of the loop, then the program will be in a state which still satisfies I after a single execution of the loop body. This can be done as before using two applications of the assignment axiom, the rule of sequential composition, and the precondition stengthening rule. Working backwards, we get the following precondition for `i := i + 1`.

$$\{s = (i+1) * i/2\}\ \texttt{i := i + 1}\ \{s = i * (i-1)/2\}$$

Using this precondition as a postcondition for `s := s + i`, we obtain the following precondition for the body of the loop.

$$\{s + i = (i+1) * i/2\}\ \texttt{s := s + i}\ \{s = (i+1) * i/2\}$$

By simplifying this precondition we get exactly I. Since $I \wedge B \Rightarrow I$, the result follows by precondition strengthening.

The final step is to show that Q is a valid postcondition after the loop terminates. For this we need to prove that $I \wedge \neg B \Rightarrow Q$. Since $\neg B$ is just $i = n$, $I \wedge \neg B$ simplifies to exactly Q, namely $s = n * (n-1)/2$.

We conclude that the loop is partially correct with respect to precondition P and postcondition Q. □

Note that in the above example we have still not proved that the while loop is (totally) correct with respect to its specifications. Indeed even if `i` and `s` both start off as zero, if `n` is negative to begin with, the loop will not terminate and therefore is not totally correct. Proving termination is the subject of the next section.

3.5 Termination

In order to show that a while statement is totally correct we need to show both that it is partially correct using the rules defined in the last section and that it terminates. For a program to be totally correct, every loop in the program must terminate. In this section, we consider how such proofs can be formalised.

One way of checking that a loop is guaranteed to terminate is to discover an integer function t of the program variables such that (1) the value of t is decremented by each execution of the loop and (2) as long as the loop has not terminated, the value of t is guaranteed to be greater than zero. Finding such a function and proving the above two results will then ensure that t must eventually be decremented to a value which is less than or equal to zero and hence that the loop must terminate. The function t is called the *bound* (or *variant*) *function*.

Given a while statement with Boolean condition B and body S along with an invariant I and bound function t, the formal statements corresponding to the above two conditions which have to be proved are as follows.

1. $I \wedge B \Rightarrow (t > 0)$
2. $\{I \wedge B\}\ t' := t;\ S\ \{t < t'\}$

The first of these shows that if the loop has not terminated (since B is still true), then the value of t must still be positive. In the second statement, t' is simply being used as a temporary variable to hold the value of t before the body of the loop executes. If the second statement can be proved valid, then each iteration of the loop decrements the value of t.

Example 3.5.1 Determining the bound function is often easy, particularly when there is an explicit loop variable which is being incremented or decremented. Consider once more the program of Example 3.4.1.

```
WHILE (i <> n) DO
  BEGIN s := s + i; i := i + 1 END
```

Here what is intended is that `i` is incremented until it becomes equal to `n`. So the bound function is simply $t = n - i$, which is the number of iterations the loop still has to perform at any stage. However, with I being $s = i*(i-1)/2$ we cannot show that $I \wedge B \Rightarrow (t > 0)$, the reason of course being that `n` could start off being less than `i`. This should alert us to the fact that something is wrong. What we need to do in order to succeed with the above proof is to strengthen the invariant to include the term $i \leq n$, that is $I \equiv (i \leq n \wedge s = i*(i-1)/2)$. This will also necessitate strengthening the original precondition P in the same way to be able to prove that $P \Rightarrow I$. This corresponds to our intuition, since the precondition now states that the loop will be totally correct only if the program starts in a state in which $i \leq n$.

Now we can prove the two conditions necessary for termination. For the first we have to show that $I \wedge B \Rightarrow (t > 0)$. Looking at the formulae for I and B, we see that $I \wedge B \Rightarrow i < n$. This in turn is equivalent to stating that $n - i > 0$. Since $n - i$ is t, we have shown that $t > 0$ as required.

The second condition requires us to prove that $\{I \wedge B\}\ t' := t;\ S\ \{t < t'\}$ is a correct specification. Using the fact that $t = n - i$ we obtain a precondition for S in the usual way.

$$\{n - (i+1) < t'\}\ \texttt{i := i + 1}\ \{n - i < t'\}$$
$$\{n - (i+1) < t'\}\ \texttt{s := s + i}\ \{n - (i+1) < t'\}$$

Using this precondition as a postcondition for $t' := t$, we obtain the precondition $\{n - (i+1) < t\}$. Substituting $n - i$ for t, we obtain a condition which is always true and therefore certainly implied by $I \wedge B$. In other words, t is decremented each time around the loop irrespective of whether $I \wedge B$ is true or not. We have thus shown that the loop always terminates given the stronger precondition P. □

We conclude this chapter with some examples of complete proofs of (total) correctness.

3.6 Correctness Proofs

We provide proofs of (total) correctness for two sample programs in this section. Once again we do not expect programmers to go through this process for each module of code they write. Current programming methodology, however, does encourage the use of pre- and postconditions for each module, as well as specification of invariants for each loop in a program. As stated previously, these provide useful documentation as well as helping the testing process if the compiler being used offers a facility for checking assertions automatically. The hope is that as program verification tools become more widely available they will use these assertions in the automatic verification of portions of code.

Example 3.6.1 Consider the following program which sets the variable `i` to the highest power of 2 that is less than or equal to the value of the variable `n`.

P : $\{n > 0\}$
```
i := 1;
WHILE (2 * i <= n) DO
  i := 2 * i;
```
Q : $\{(0 < i \leq n < 2 * i) \wedge \exists(p :: i = 2^p)\}$

There are a number of steps involved in proving the above program correct with respect to P and Q.

1. Find an invariant I for the loop: a correct invariant is

$$I \equiv (0 < i \leq n \ \wedge \ \exists(p :: i = 2^p)).$$

   Also find a bound function t for the loop: $n - i$ will suffice.

2. Show that I is true before the loop, that is, $\{P\}$ `i := 1` $\{I\}$ is valid. Substituting 1 for i in I we get

$$\begin{aligned} I(1) &\equiv 0 < 1 \leq n \ \wedge \ \exists(p :: 1 = 2^p) \\ &\equiv 1 \leq n \ \wedge \ true \\ &\equiv n > 0 \\ &\equiv P \end{aligned}$$

3. Show that I is an invariant, that is, $\{I \wedge B\}$ `i := 2 * i` $\{I\}$. First find a precondition P_1 for `i := 2 * i`. By the assignment axiom, this is

$$P_1 \equiv (0 < 2 * i \leq n \ \wedge \ \exists(p :: 2 * i = 2^p)).$$

   Now if we can show that $I \wedge B \Rightarrow P_1$, then the result follows by the precondition strengthening rule. For $I \wedge B$ we get

$$0 < i \leq n \ \wedge \ \exists(p :: i = 2^p) \ \wedge \ 2 * i \leq n.$$

   Combining the first and last terms while noting that if there is a p satisfying $i = 2^p$ then there is a p satisfying $2 * i = 2^p$ (namely one more than the first p), we can see that this does imply precondition P_1.

4. Show that, on termination of the loop, the program will be in a state satisfying Q, that is, prove that $I \wedge \neg B \Rightarrow Q$. In fact, $I \wedge \neg B$ gives us a formula equivalent to Q; namely

$$0 < i \leq n \ \wedge \ \exists(p :: i = 2^p) \ \wedge \ 2 * i > n.$$

5. Show that the bound function $t = n - i$ is positive whenever the body of the loop is executed, that is, $I \wedge B \Rightarrow t > 0$.

$$\begin{aligned} I \wedge B &\equiv 0 < i \leq n \wedge \exists(p :: 1 = 2^p) \wedge 2 * i \leq n \\ &\Rightarrow 0 < i \leq n \wedge 2 * i \leq n \\ &\Rightarrow n - i > 0 \end{aligned}$$

6. Show that each loop iteration decreases t, that is, prove that

$$\{I \wedge B\}\ \texttt{t' := n - i; i := 2 * i}\ \{n - i < t'\}$$

is a valid specification. Applying the assignment axiom to `i := 2 * i`, we get the precondition $\{n - (2 * i) < t'\}$. Now applying the assignment axiom to `t' := n - i`, we get precondition $\{n - (2 * i) < n - i\}$ which is exactly $\{i > 0\}$. It is clear that $I \wedge B \Rightarrow i > 0$, so the result follows by the rule of sequential composition and precondition strengthening.

Steps (1) to (4) show that the program is partially correct with respect to P and Q, while steps (5) and (6) show that the program terminates. □

Example 3.6.2 Now consider the following program which finds a value for variable `k` such that `a[k]` is the maximum value of the array `a[0:n-1]`.

P: $\{n > 0\}$
```
i := 1; k := 0;
WHILE (i < n) DO
  BEGIN
    IF a[i] > a[k] THEN k := i;
    i := i + 1;
  END
```
Q: $\{\forall(j : 0 \leq j < n : a[k] \geq a[j])\}$

Rather than referring to each of the assertions and statements of the program explicitly all the time, let us give them names as follows.

$\{P\}$
S_1; S_2;
WHILE B_1 DO
  BEGIN
    IF B_2 THEN S_3;
    S_4;
  END
$\{Q\}$

As before, there are a number of steps to follow. We will prove some of them, leaving the others as exercises.

1. An invariant I for the loop is

$$0 < i \leq n \wedge \forall(j : 0 \leq j < i : a[k] \geq a[j])$$

which says that `k` is the array index of the largest element found so far (that is, between `a[0]` and `a[i-1]`). The bound function t is $n - i$ as before.

2. Prove that I is true before the loop executes, that is, prove that $\{P\}\ S_1;\ S_2;\ \{I\}$ is correct. We leave this as a straightforward exercise.

3. Prove that I is an invariant of the loop, that is, prove that

$$\{I \wedge B_1\} \texttt{ IF } B_2 \texttt{ THEN } S_3; S_4 \{I\}$$

is correct. This step itself is best broken down into two steps: (a) find a precondition P_1 for S_4, and (b) show that $\{I \wedge B_1\}$ IF B_2 THEN S_3 $\{P_1\}$ is correct.

(a) Precondition P_1 is obtained by substituting $i+1$ for i in I (according to the assignment axiom) which yields

$$P_1 \equiv (0 < i+1 \leq n \wedge \forall(j : 0 \leq j < i+1 : a[k] \geq a[j])).$$

(b) According to the proof rule for one-sided conditionals, we need to prove (i) $I \wedge B_1 \wedge \neg B_2 \Rightarrow P_1$, and (ii) $\{I \wedge B_1 \wedge B_2\}$ k := i $\{P_1\}$.

i. The proof proceeds as follows:

$$\begin{aligned} I \wedge B_1 \wedge \neg B_2 &\equiv 0 < i \leq n \wedge \forall(j : 0 \leq j < i : a[k] \geq a[j]) \wedge i < n \\ &\quad \wedge a[i] \leq a[k] \\ &\equiv 0 < i+1 \leq n \wedge \forall(j : 0 \leq j < i : a[k] \geq a[j]) \\ &\quad \wedge a[k] \geq a[i] \\ &\equiv P_1 \end{aligned}$$

ii. First, find a precondition P_2 for k := i using postcondition P_1 and the assignment axiom, that is, substitute i for k in P_1 yielding

$$P_2 \equiv (0 < i+1 \leq n \wedge \forall(j : 0 \leq j < i+1 : a[i] \geq a[j])).$$

Next, show that $I \wedge B_1 \wedge B_2 \Rightarrow P_2$.

$$\begin{aligned} I \wedge B_1 \wedge B_2 &\equiv 0 < i \leq n \wedge \forall(j : 0 \leq j < i : a[k] \geq a[j]) \wedge i < n \\ &\quad \wedge a[i] > a[k] \\ &\equiv 0 < i+1 \leq n \wedge \forall(j : 0 \leq j < i+1 : a[i] \geq a[j]) \\ &\equiv P_2 \end{aligned}$$

We conclude that I is an invariant.

4. Prove that Q is true after the loop terminates, that is, prove that $I \wedge \neg B_1 \Rightarrow Q$.

$$\begin{aligned} I \wedge \neg B_1 &\equiv 0 < i \leq n \wedge \forall(j : 0 \leq j < i : a[k] \geq a[j]) \wedge i \geq n \\ &\equiv 0 < i = n \wedge \forall(j : 0 \leq j < n : a[k] \geq a[j]) \\ &\Rightarrow \forall(j : 0 \leq j < n : a[k] \geq a[j]) \\ &\equiv Q \end{aligned}$$

The above four steps have shown that the given program is partially correct with respect to precondition P and postcondition Q. We leave the two steps necessary to show termination as an exercise. □

Bibliography

[1] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, Vol. 12, No. 10 (Oct. 1969), pp. 576–580.

In this paper, the idea of using the rules of inference we have covered was proposed for the first time. Of course, much work has been done since then, resulting in a host of papers on the subject, but this paper does not appear dated and remainsvery readable.

[2] J.H. Fetzer, "Program Verification: The Very Idea," *Commun. ACM*, Vol. 31, No. 9 (Sept. 1988), pp. 1048–1063.

This is a recent rebuttal by a philosopher of the whole concept of program verification. Although it is somewhat difficult to follow some of the arguments, this paper makes fascinating reading and has provoked a flurry of responses in subsequent issues of *Communications of the ACM*.

[3] R.C. Backhouse, "*Program Construction and Verification*," Prentice-Hall, 1986.

The pace of this book is leisurely, with plenty of examples and explanations. The first few chapters essentially cover the same material as we have, after which the idea of constructing programs in such a way that they are guaranteed to be correct is explored.

[4] D. Gries, "*The Science of Programming*," Springer-Verlag, 1981.

This book is devoted to advocating a scientific, in particular mathematical, approach to programming. A comprehensive methodology for developing correct programs is presented.

[5] M.J.C. Gordon, "*Programming Language Theory and its Implementation*," Prentice-Hall, 1988, Part I.

This is another very accessible book which covers, in Part I, much the same material we have done. The rest of the book is devoted to a description of λ-calculus, followed by the development of an actual program verifier written in Lisp.

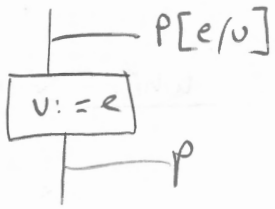
[6] J.C. Reynolds, "*The Craft of Programming*," Prentice-Hall, 1981.

This book covers the preceding material at a much faster pace than we have done, after which it treats more general and advanced program constructs.

[7] E.W. Dijkstra, "*A Discipline of Programming*," Prentice-Hall, 1976.

This remains a classic book in the area of programming methodology and program verification.

Assignment: $\{P[e/v]\}\ v := e\ \{P\}$



Precondition strengthening:

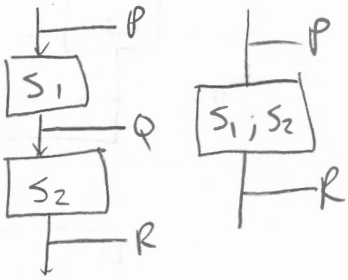
$$\frac{P \Rightarrow P' \qquad \{P'\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

Postcondition weakening:

$$\frac{\{P\}\ S\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

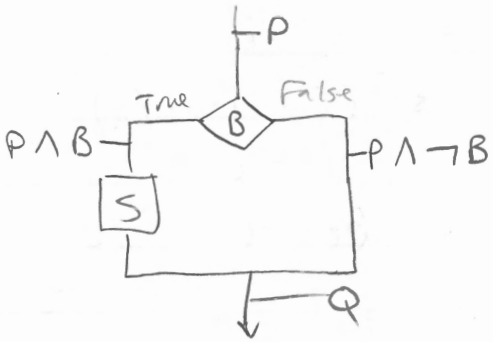
Sequential Composition:

$$\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1\ ;\ S_2\ \{R\}}$$



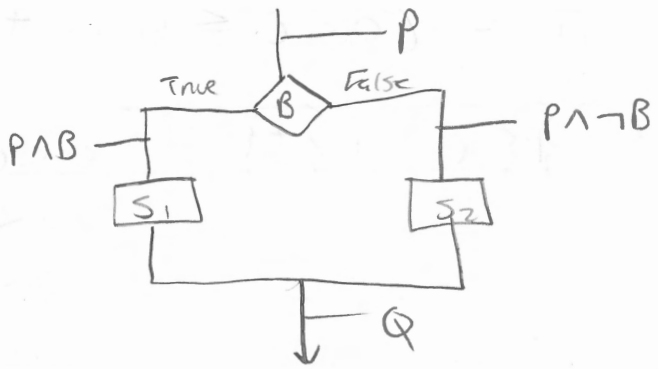
One sided conditionals

$$\frac{\{P \wedge B\}\ S\ \{Q\} \qquad P \wedge \neg B \Rightarrow Q}{\{P\}\ \text{IF } B \text{ THEN } S\ \{Q\}}$$



Two sided conditionals:

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \qquad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2\ \{Q\}}$$

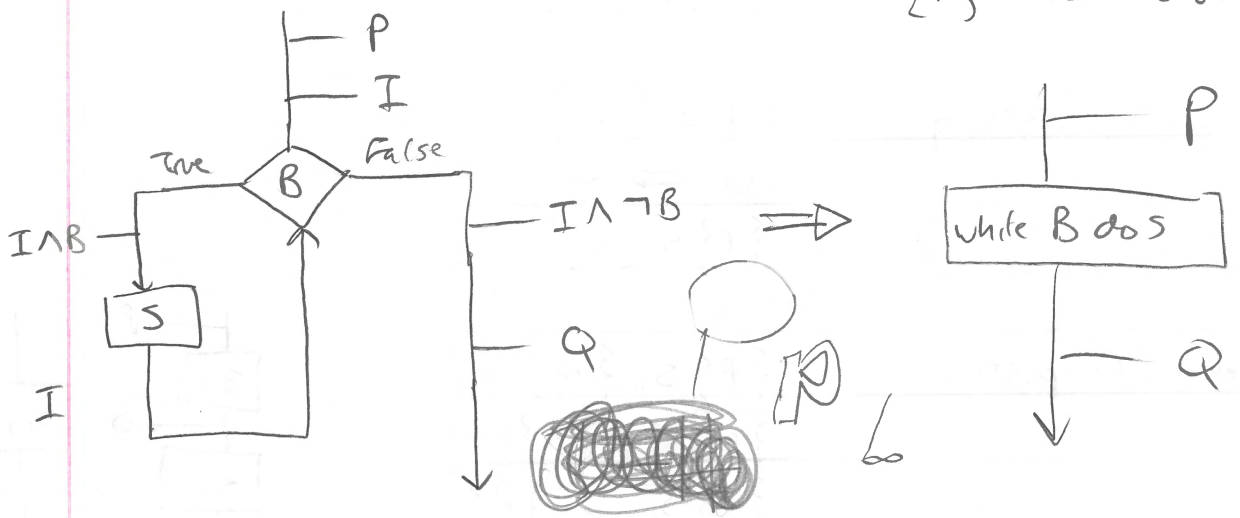


While Statements

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{WHILE } B \text{ DO } S\ \{I \wedge \neg B\}}$$

Derived Proof Rule

$$\frac{P \Rightarrow I \qquad \{I \wedge B\}\ S\ \{I\} \qquad \{I \wedge \neg B\} \Rightarrow Q}{\{P\}\ \text{WHILE } B \text{ DO } S\ \{Q\}}$$



$P = \{i = 0 \wedge s = 0\}$ $\quad Q = \{s = n*(n-1)/2\}$

$I = i*(i-1)/2$

$I = \quad s = s + i$

$s = (i+1)*(i/2)$

$i = i + 1$

$s = i*(i-1)/2$

$s+i = (i+1)*i/2$

$= i+1 \times i + 2i/2$

$= \frac{3i^2 + 3i}{2}$

$s+i = (i+1)*i/2$

$= \frac{i+i}{i} = 2i/2 = (i+1)$

$P \quad \{n > 0\}$

$Q \quad \{0 < i \leq n < 2*i\} \wedge \exists (p :: i = 2^p)$

$I = (0 < i \leq n \wedge \exists (p :: i = 2^p$

$\{P\}\ i = 1\ \{I\}$ $\quad 0 < i \leq n \wedge \exists (p :: i = 2^p)$

$0 < 1 \leq n \wedge \exists (p :: 1 = 2^p)$ $\quad p = 0$

$0 < 1 \leq n \wedge true$

$1 \leq n \quad n > 0$ $\quad P$

$I \wedge B \ S \ I \quad 0 < 2*i \leq n < 4*i$

$i := 2*i$

$(0 < i \leq n < 2*i) \wedge \exists (p :: i = 2^p)$

$\{n \geq 0\} = P$

```
y:=1 ; z:=x ; k:=n;
while (k!=0) do
    begin
        k:=k-1 ; y:=y*z
    end
```

$\{y = x^n\} = Q$

Show that $(k \geq 0 \wedge x^n = y * z^k)$ is a loop invariant.

$y := y * z$

$\{k \geq 0 \wedge x^n = (y*z) * z^k\}$

$k := k - 1$

$\{(k-1) \geq 0 \wedge x^n = (y*z) * z^{k-1}\}$

need to show $I \wedge B \Rightarrow \{k > 0 \wedge x^n = (y * z^k)\}$

Show $(I \wedge \neg B \Rightarrow Q)$

$k \geq 0 \quad x^n = y * z^k \wedge \underbrace{k = 0}_{\neg B}$

$k = 0 \wedge x^n = y \qquad Q = y = x^n$

$\Downarrow$

Q